一、实验目的与要求

实验目的：

1．熟悉博弈树及博弈搜索；

2．掌握 minmax 搜索算法和 alpha-beta 剪枝算法；

3．掌握评估函数设计方法，运用博弈搜索解决吃豆人游戏问题；

实验要求：

1. 实验提交文件为实验报告和相关程序代码，以压缩包的形式提交，命名规则为“学号数字+姓名+Task2”，如 2020154099 张三 Task2；

2. 所有素材和参考材料需列明出处，实验报告中的图片和程序代码建议标注个人水印或标识信息：姓名，班级，学号信息；

二、实验内容与方法

实验内容：

1. 改进 Reflex 智能体；

2. 设计 Minimax 智能体；

具体细节参见文件“吃豆人指引.doc”

三、实验步骤与过程

本次实验报告的内容思维导图如下所示，感谢阅读，审批辛苦了！

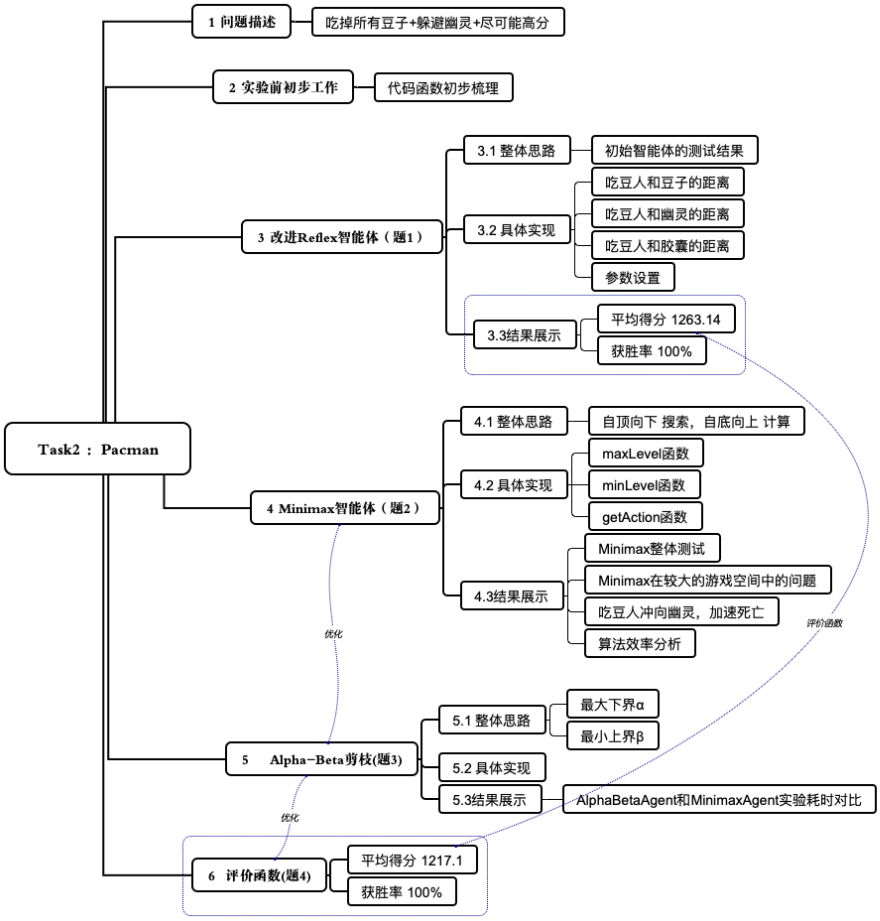

图 1:本次实验报告内容思维导图

1 问题描述

吃豆人游戏实际上是吃豆人与幽灵之间的博弈游戏，吃豆人根据豆子和幽灵的情况不断

地调整自己的行动，最后完全吃完地图上的豆子，避开幽灵的攻击，这样就赢得了胜利。所以我们的目标是：[1]

- **吃掉所有豆子**
- **躲避幽灵**
- **尽可能高分**

2 实验前初步工作

首先体验一下经典的吃豆人游戏！（调用代码：*python pacman.py*）在这种情况下，我们没有使用智能体，而是完全依靠键盘来操作吃豆人，得分也完全依赖我们对游戏的操作能力。这让我更想设计一个“外挂（智能体）”了…

图 2：吃豆人初体验

通过对实验压缩包的阅读，可以发现，本次实验需要进行修改的是 mutiAgents.py 文件的 **ReflexAgent 类、MinimaxAgent 类和 AlphaBetaAgent 类**，因为整个项目工程量比较大，我在文件的注释帮助下找到这些类在 pacman.py、util.py、game.py 文件中的实现方式，并进行**初步梳理**，如图 3 所示：

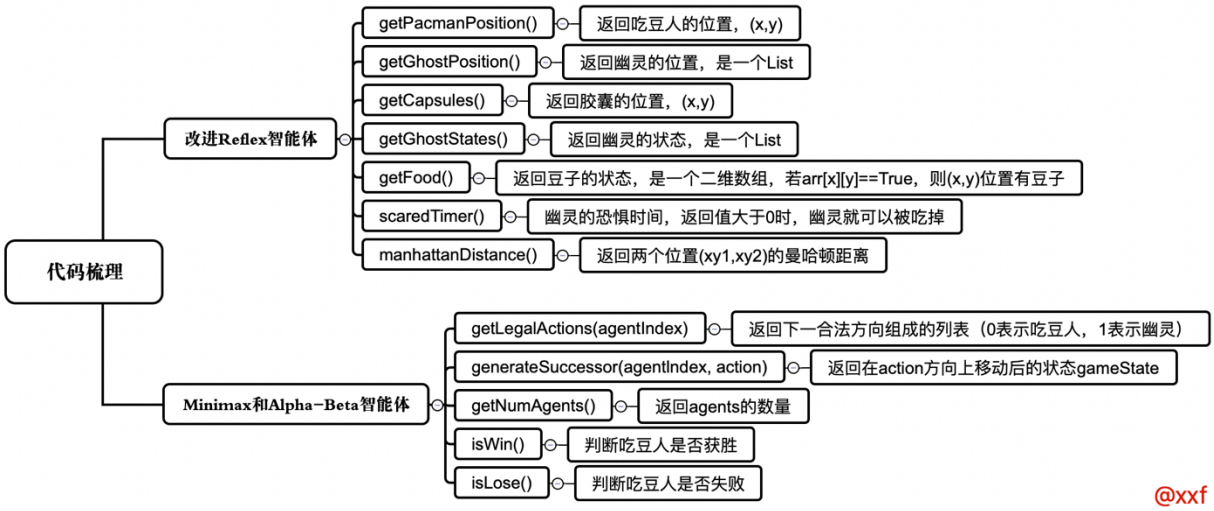

图 3 代码梳理思维导图

3 改进 Reflex 智能体（题 1）

3.1 整体思路

ReflexAgent 智能体要可以高效吃掉界面上的所有豆子，同时还能避免与幽灵相遇。multiAgent.py 文件中已经提供了一个 ReflexAgent 的框架，在未进行改进的情况下，我们使

[1] 参考：https://blog.csdn.net/Pericles_HAT/article/details/116901139

用 *python pacman.py -p ReflexAgent --frameTime 0 -k 2 -l openClassic -n 100 -q -g DirectionalGhost -f* 语句**对初始的智能体进行 100 次测试**。结果（图 4）发现，当使用了命令 *-g DirectionalGhost* 设置使得游戏中的**幽灵更聪明和有方向性时**，这个初始智能体毫无应对之力，平均得分为-302.77，一直都是 Loss，获胜的几率为 0！

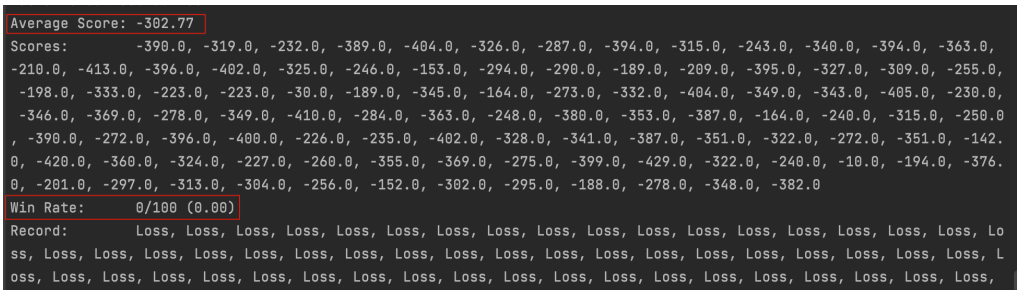

图 4 初始智能体的测试结果

因此，我们要对 multiAgent.py 文件中 ReflexAgent 智能体进行改进。首先**观察 ReflexAgent 类的结构**，evaluationFunction 函数的输入参数为当前 GameState 和下一步 Action，返回值应当为经过 Action 后的状态的评估值。而 getAction 函数计算出某个 GameState 经过所有可采取的 Action 后的评估值数组，并选择出评估值最大的状态。

```
1.  class ReflexAgent(Agent):
2.      def getAction(self,gameState)
3.      def evaluationFunction(self, currentGameState, action)
```

发现决定 ReflexAgents 表现的主要是类中的 evaluationFunction 函数，而如何让吃豆人在避开幽灵的同时更快地吃到豆子，是 evaluationFunction 函数设置的关键。充分利用 pcaman.py 中的 GameState 类中的函数和 util.py 中的 manhattanDistance 函数后，我们在评估函数 evaluationFunction 设计中添加了三种距离进行评判，分别是**吃豆人和豆子的距离、吃豆人和幽灵的距离、吃豆人和胶囊的距离**，需要注意的是这里距离特征使用的是**曼哈顿距离的倒数**，然后将这三个距离进行组合相加得到评估值，这里重点是如何设置这**三个距离的权重**，使得整个评估函数是最优的。[2]

3.2 具体实现

（1）吃豆人和豆子的距离

通过遍历 newFood 二维数组，计算新位置与所有豆子的**距离并取倒数累加起来并乘以权重系数 w[0]**，作为新位置的得分。这样吃豆人每次都选择离豆子较近的方向前进，避免了盲目搜索，提高了效率。

```
1.  # 吃豆人和豆子的距离 （1/曼哈顿距离 mdis）
2.  FoodDis = 0
3.  for x in range(newFood.width): #遍历二维数组
4.      for y in range(newFood.height):
5.          if newFood[x][y]:
6.              FoodDis += w[0]*(1/manhattanDistance(newPos, [x,y]))
```

（2）吃豆人和幽灵的距离

因为只有当幽灵靠近时，吃豆人才会受到威胁，所以**无需考虑距离吃豆人较远的幽灵**，所以在这里我设置了一个吃豆人**要开始躲避幽灵的最大距离范围 alpha**，当两者距离

[2] 参考：https://blog.csdn.net/weixin_34138521/article/details/94193368

小于 Alpha 时，评估函数才开始将幽灵的存在考虑进评估值函数内。首先遍历 newGhostPos 和 newScaredTimes，通过调用 manhattanDistance 函数计算新位置与幽灵的曼哈顿距离 mdis。然后根据两者的位置可以分为以下三种情况：

A:当幽灵处于**恐惧状态**时，应该让吃豆人尽力去吃掉幽灵，此时令评估函数**加上** w[1]*(1/mdis)；

B:当幽灵处于**正常状态**且到吃豆人新状态的曼哈顿距离在**区间[1,alpha]内**时，吃豆人需要尽可能远离幽灵，因此令评估函数**减去** w[2]*(1/mdis)；

C:当幽灵距离**吃豆人新状态为 0** 时，要让吃豆人不要走到这个位置，所以令评估函数**减去一个很大的值**（eg:100000），以降低吃豆人被吃掉的风险。

```
# 吃豆人和幽灵的距离 （1/曼哈顿距离 mdis）
 GhostDis = 0
 for ghosts,scareTimes in zip(newGhostPos, newScaredTimes):
    mdis = manhattanDistance(newPos, ghosts)
    if scareTimes > 0: # 恐惧状态
        GhostDis += w[1]*(1/mdis)
    elif 0 < mdis and mdis < alpha: # 正常状态
        GhostDis -= w[2]*(1/mdis)
    elif mdis==0: # 下一步可能是幽灵
        GhostDis -= 100000
```

（3）吃豆人和胶囊的距离

通过遍历所有胶囊的位置 CapsulesPos，每次计算出胶囊与吃豆人的曼哈顿距离 mdis，然后分为两种情况：

A：当 **mdis > 0** 时，需要提高吃豆人吃到胶囊的概率，因此评价得分**加上** w[3]*(1/mdis)；

B：当 **mdis = 0** 时，吃豆人应该直接去吃胶囊，所以评价得分**加上一个很大的值**（eg:100000），以确保吃豆人进行吃胶囊这一步骤。

```
# 吃豆人和胶囊的距离 （1/曼哈顿距离 mdis）
CapsulesDis = 0
for capsules in CapsulesPos:
    mdis = manhattanDistance(newPos, capsules)
    if mdis > 0: #尽量去吃
        CapsulesDis += w[3]*(1/mdis)
    elif mdis == 0: # 下一步可能是胶囊
        CapsulesDis += 100000
```

（4）参数设置

在改进 Reflex 智能体的过程中，针对每个距离我一共设置了四个特征权重 w[0],w[1],w[2],w[3]以及一个参数 Alpha（见表 1），但**参数如何取值**才能使得评估函数最优成了一个**难题**。

首先查阅文献，发现通过利用强化学习解决吃豆人问题的结果都差别不大，以 Gnanasekaran, A,2017[3]为例，其通过**消融分析（Ablation Analysis**，即逐步去掉各种特征，分析平均得分和获胜率）研究方法表明（见图 5），在获胜率一定的情况下，吃豆人距离豆子的最短距离、距离害怕状态的幽灵的距离、距离胶囊的距离对游戏平均得分的影响是最大的（因为去掉这些特征后，平均得分下降最多）。同时，如果为了确保获胜率，吃豆人距离正常状态的幽灵的距离的权重必需足够高。

Table I: Ablation Analysis

Component	**Avg. Score**	**Win Rate**
Overall System	1608	92%
min. distance scared ghost	1581	90.2%
inv. min. distance active ghost	1583	91.6%
min. distance active ghost	1594	91.8%
min. distance capsule	1591	90.2%
no. scared ghosts 2 steps away	**1545**	92.2%
no. scared ghosts 1 step away	**1438**	91.2%
distance to closest food	**1397**	90%

图 5 吃豆人问题对特征的消融分析（Gnanasekaran, A,2017）

有了文献帮助的初步判断后，我再不断**迭代调参**，最后为了极大的降低吃豆人死亡的可能性，先将 w[2]调至为最大权重，接着，追踪恐惧状态的幽灵的权重较追踪豆子更高，因此赋予 w[1]第二大的权重，对于豆子和胶囊的权重，要先吃距离最近的豆子，所以设置第三大权重为豆子 w[0], Alpha 作为开始躲避幽灵的距离试探了 1~10 后选择了最佳的 4。经过多次反复实验，最终的距离特征权重和参数如下：

表 1 距离特征权重和参数表

特征权重&参数	含义	最终取值
w[0]	吃豆子的权重	**2**
w[1]	吃恐惧状态的幽灵的权重	**6**
w[2]	躲避正常状态的幽灵的权重	**10**
w[3]	吃胶囊的权重	**0.5**
Alpha	需要开始躲避幽灵的最大距离	**4**

3.3 结果展示

使用命令行 *python pacman.py -p ReflexAgent -l openClassic -n 50 -q* 进行测试，综合考虑了三种距离，**平均分达到了 1263.14，而且获胜率是 100%**，说明评价函数设计的还不错！

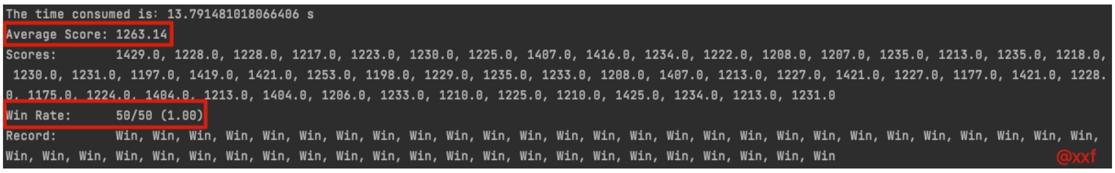
```
The time consumed is: 13.791481018066406 s
Average Score: 1263.14
Scores:         1429.0, 1228.0, 1228.0, 1217.0, 1223.0, 1230.0, 1225.0, 1407.0, 1416.0, 1234.0, 1222.0, 1208.0, 1207.0, 1235.0, 1213.0, 1235.0, 1218.0,
 1230.0, 1231.0, 1197.0, 1419.0, 1421.0, 1253.0, 1198.0, 1229.0, 1235.0, 1233.0, 1208.0, 1407.0, 1213.0, 1227.0, 1421.0, 1227.0, 1177.0, 1421.0, 1228.
0, 1175.0, 1224.0, 1404.0, 1213.0, 1404.0, 1206.0, 1233.0, 1210.0, 1225.0, 1210.0, 1425.0, 1234.0, 1213.0, 1231.0
Win Rate:       50/50 (1.00)
Record:         Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win,
Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```


图 6 ReflexAgent 测试结果

4 Minimax 智能体（题 2）

4.1 整体思路[4]

上面吃豆人游戏中，吃豆人的位置由我们自己手动控制，或者运用改进的 Reflex 智能体进行控制，现在我们需要设计出来一个 Minimax 智能体来真正**让机器与机器进行对抗**，

[3] Gnanasekaran, A., Faba, J. F., & An, J. (2017). Reinforcement Learning in Pacman. *See nalso URL http://cs229. stanford. edu/proj2017/final-reports/5241109. pdf.*

[4] https://blog.csdn.net/weixin_44662636/article/details/120340824

也就是让计算机控制吃豆人的同时，也控制幽灵，两者之间进行**零和博弈**（即一方胜利代表着另一方的失败），所以可以使用 **Minimax 算法找到最优步骤**。

大致思想是最大化自己的效益，最小化另一玩家的效益，在本实验中也就是**最大化吃豆人的效益，最小化幽灵的效益**。Minimax 算法的**伪代码**如下所示，以**自顶向下**的方式**搜索**，以**自底向上**的方式**计算**吃豆人和幽灵最佳得分下的状态。

```
# minimax 搜索伪代码
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v,min-value(successor))
    return v
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v,max-value(successor))
    return v

#minimax 计算伪代码
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is MIN: return min-value(state)
```

要使 Minimax 智能体能应对多个幽灵，在对搜索层进行设计时，要含有一个 max 层(maxLevel,吃豆人)和多个 min 层(minLevel, 幽灵)，每个 min 层对应一个幽灵。这里在搜索时有两种设计方法，以两个幽灵为例：

A：只选择**选择距离最近的幽灵放进博弈树**，在实现过程中只需**调用评估函数**，向上返回评价分数。

B：将这**两个幽灵分别放入两层**，在实现过程中，第一个幽灵会从第二个幽灵的 min 层选出最小值，第二个幽灵会从吃豆人的 max 层选出最小值。函数的**递归调用由深度和幽灵 agentIndex 控制**：吃豆人调用最小 agentIndex 的幽灵，小 agentIndex 的幽灵调用大 agentIndex 的幽灵，而最大 agentIndex 的幽灵调用吃豆人，即下一个搜索层的 max 层。

本实验使用方法 B 进行实现。

4.2 具体实现[5]

（1）maxLevel

max 层仅用于吃豆人（当 agentIndex 为 0 时表示吃豆人的移动，对应 Max 层结点；当 agentIndex>0 时表示幽灵的移动，对应 Min 层结点），传入 maxLevel 函数的参数为游戏状态和深度。maxLevel 函数流程和核心代码如下：

表 2：maxLevel 流程表

步骤	操作
1	调用函数时，更新深度 currDepth，吃豆人当前深度+1；

[5] 参考： https://blog.csdn.net/weixin_34138521/article/details/94193368

2	判断当前的状态是否达到了停止条件，即吃豆人获胜，吃豆人失败或是深度达到设定深度，并初始化 maxvalue 为负无穷。
3	用 actions 记录合法的移动，再根据当前游戏状态和合法移动生成后继状态，每生成一个后继状态，maxvalue 与来自后继状态即 mini 层的值来做计较，取最大值；
4	最后返回该层的 maxvalue。

```
# max 层仅用于吃豆人，吃豆人的 agentIndex =0
def maxLevel(gameState, depth):  # 传入游戏状态，深度
    currDepth = depth + 1  # 吃豆人当前深度+1
    # 判断当前状态是否达到了停止条件
    if gameState.isWin() or gameState.isLose() or currDepth == self.depth:
        return self.evaluationFunction(gameState)
    maxvalue = -999999  # 初始化 maxvalue 为负无穷
    actions = gameState.getLegalActions(0)  # actions 记录合法的移动
    # 根据移动来生成后继状态
    for action in actions:
        successor = gameState.generateSuccessor(0, action)
            # 用 maxvalue 与 min 层比较，取最大值
        maxvalue = max(maxvalue, minLevel(successor, currDepth, 1))
    return maxvalue  # 在该层中找到最大值传导到上一层
```

（2）minLevel

minLevel 是对幽灵而言的，传入的参数为游戏状态，深度和 agentIndex。流程同 maxLevel 流程类似，不同的是，如果**只有 1 个幽灵**，当它获取后续 Action 并要寻找下一步的最小值时，其会从 Max 层中选出评估值最小的状态节点；如果**有 2 个幽灵**，博弈树就会为 2 个幽灵设定决策层，第 1 个幽灵会从第二个幽灵的 Min 层选出最小值，第 2 个幽灵会从吃豆人的 max 层选出最小值。minLevel 的函数流程和核心代码如下所示：

表 3：minLevel 流程表

步骤	操作
1	初始化 minvalue 为正无穷，然后判断当前状态是否处于赢或者输的状态，如果是则退出；
2	获取幽灵的合法行动，传入 agentIndex，根据幽灵的移动生成新的后继状态；
3	根据幽灵的个数，用 agentIndex 来判断，如果是最后一个幽灵，则将，minvalue 与来自 max 层的值来做对比，取其最小值；如果不是最后一个幽灵，则与下一个 min 层做对比，取其最小值；
4	最后，返回 minvalue。

```
# min 层是对于幽灵而言的，参数为游戏状态，深度，agentIndex
    def minLevel(gameState, depth, agentIndex):
        minvalue = 999999  # 初始化 minvalue 为正无穷
        if gameState.isWin() or gameState.isLose():  # 赢或者输了就退出
            return self.evaluationFunction(gameState)
        # 获取当前状态合法的移动
        actions = gameState.getLegalActions(agentIndex)
        for action in actions:
```

```
                # 根据移动生成新的后继状态
            successor = gameState.generateSuccessor(agentIndex, action)
            # 根据幽灵的个数，找到 min 层中的最小值 minvalue
            if agentIndex == (gameState.getNumAgents() - 1):#最后一个幽灵
                minvalue = min(minvalue, maxLevel(successor, depth))
            else: #不是最后一个幽灵
                minvalue = min(minvalue, minLevel(successor, depth, agentIndex + 1))
        return minvalue
```

（3）getAction 函数

设计好 maxLevel 和 minLevel 后，还需要进行遍历。首先初始化 currentScore 和 returnAction，并获取吃豆人当前状态的合法行动。然后由当前状态遍历，不断生成后继状态，因为下一层是 min 层，所以可以**把当前的这一层看成是 min 层的根结点（深度为 0）**，然后选择后继状态的最大的值，同时更新其移动的方向，最后返回的是最高得分移动的方向。

```
currentScore = -999999 # 初始化
        returnAction = ''
        # 获取吃豆人合法的行动
        actions = gameState.getLegalActions(0)
        for action in actions:
            nextState = gameState.generateSuccessor(0, action)
            # 接下里的一层是 min 层，可以把当前的这一层看作 min 层的根结点
            score = minLevel(nextState, 0, 1)
            # 选择后继状态中最大值移动
            if score > currentScore:
                returnAction = action
                currentScore = score
        return returnAction  # 返回最高得分移动的方向
```

4.3 结果展示

（1）细节：输出代码运行时间

为了更好地进行结果展示和算法评价，我在 pacman.py 中增加了**输出运行时间**的代码，只要将下面的语句放在 runGames 函数的相应位置，就可以实现了。

```
import time
start = time.time()
…
end = time.time()
print("The time consumed is：" + str(end - start) + " s") #运行时间（秒）
```

（2）选择幽灵的个数

实际运行游戏时候，可以用**参数-k (<3)来选择幽灵个数**。当 k=1 时，只有一个幽灵，语句为 *python pacman.py -p MinimaxAgent -a depth=1 -k 1*；当应对两个幽灵时，语句为 *python pacman.py -p MinimaxAgent -a depth=1 -k 2*，页面分别如下所示。

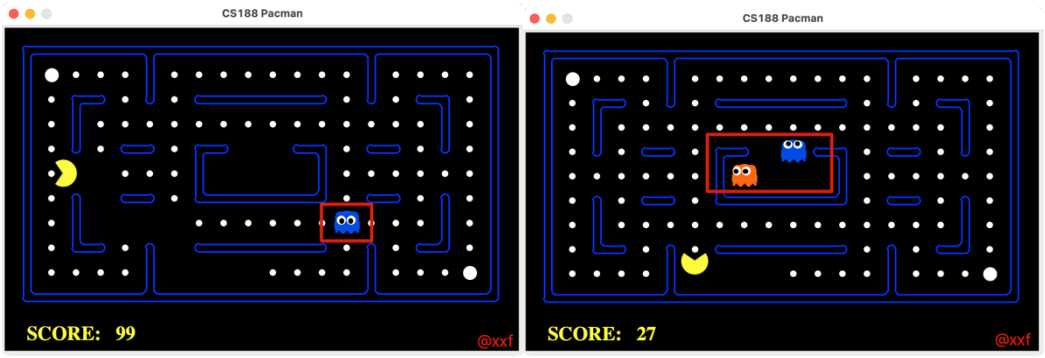

图 7 使用参数-k 设置幽灵个数（左：k=1，右：k=2）

（3）Minimax 整体测试

在 **minimaxClassic 布局**游戏的初始状态下，Minimax 树的深度为 1、2、3、4 的结点数值分为 9、8、7、-492，尽管深度为 4 的 Minimax 树的预测结果极差，但本实验 minimax 智能体测试下胜率为 **64%**，使用的命令为 *python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4 --frameTime 0 -n 100 -q*

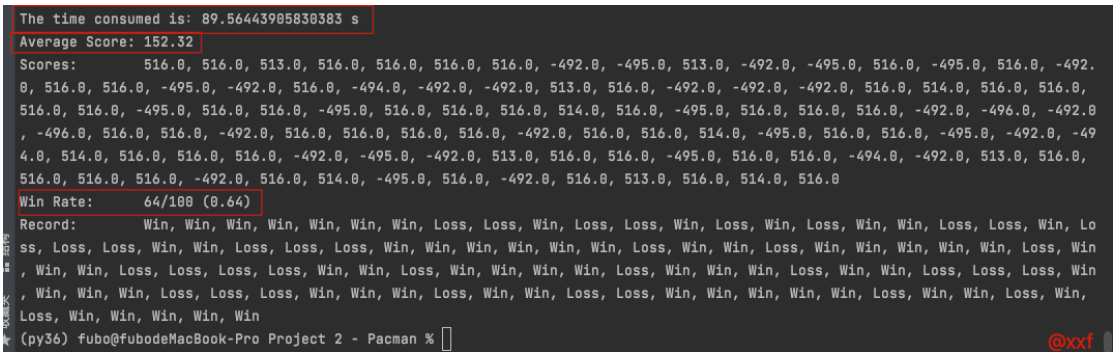

图 8 Minimax 整体测试结果（胜率为 64%）

（4）Minimax 在较大的游戏空间中的问题

在 openClassic 和默认的 mediumClassic 的**更大的游戏空间中**，吃豆人很难死亡，同时也很难取胜，常常在一个区域内翻来覆去。通过命令 *python pacman.py -p MinimaxAgent -l openClassic* 测试。正如下图 9 所示，吃豆人在图示位置**停止不动了**，不去吃下一个豆子了。而且每当只剩最后一个豆子时，吃豆人总是在该点周围翻来覆去，但不吃那个点，其主要原因是吃豆人在当前搜索深度下，**不知道吃掉了那个点之后要怎么走**。据多次试验观察，吃豆人往往需要在被幽灵逼迫的情况下才会去掉最后省下的那个豆子赢得胜利。

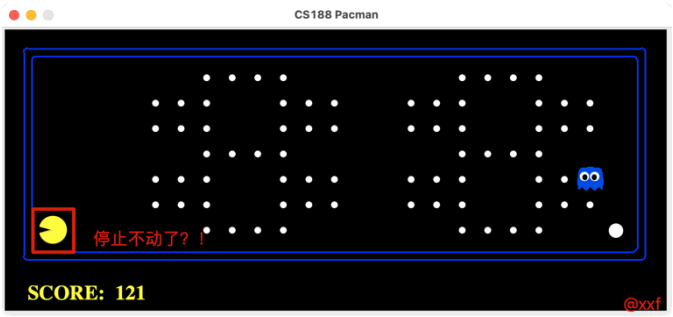

图 9 Minimax 在较大的游戏空间中的停止不动了

（5）吃豆人冲向幽灵，加速死亡

值得注意的是，**在 trappedClassic 界面**中，当吃豆人发现它将不可避免的死亡时，利用 **minimax 算法**，尽管最后的状态评估值都小于 0，但吃豆人也会寻找一个最大的评估值，即会冲向幽灵**加速死亡，结束游戏**，这是因为**不断存活也会继续扣分**。（语句：*python pacman.py -p MinimaxAgent -l trappedClassic -a depth=3*）再做一下对比，发现如果是 **ReflexAgent**，它

会**继续吃豆子，直到死亡**。（语句：*python pacman.py -p ReflexAgent -l trappedClassic*）

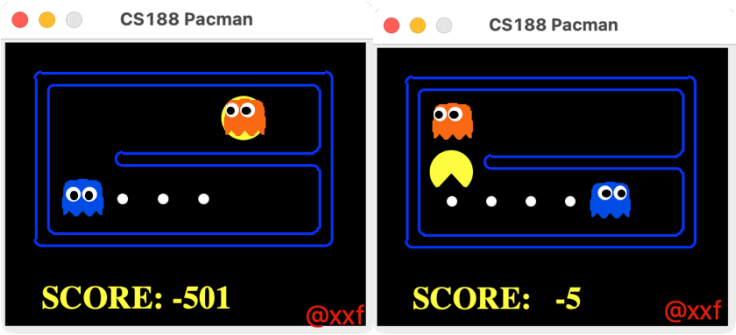

图 10 MinimaxAgent（左）和 ReflexAgent（右）在必死无疑的情况下的对比

（6）算法效率分析

除此之外，由于 minimax 算法的**时间复杂度达到 O(b^m)**以及**空间复杂度达到 O(bm)**，其搜索空间巨大，获得精确的解完全不可能。当开始运行的时候，以语句：*python pacman.py -p MinimaxAgent -k 2 -a depth=4* 为例，**运行时间高达 55.27 秒**。因此我们需要**引入 Alpha-Beta 剪枝策略。**

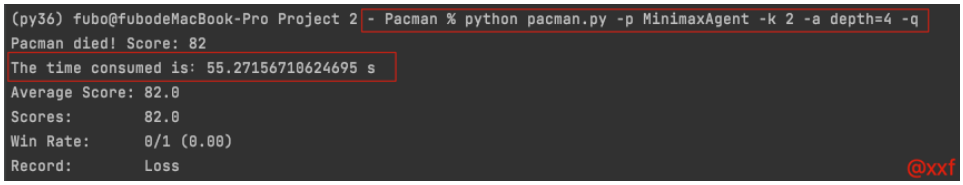

图 11 MinimaxAgent 算法运行时间高举例

5 Alpha-Beta 剪枝(题 3)

5.1 整体思路

在 Minimax 算法中，**有一些状态实际上是不需要我们去遍历的**，这会大大增加算法的时间复杂度，所以需要使用 Alpha-Beta 剪枝算法来进行改进。Alpha-Beta 剪枝是在 Minimax 算法的基础上进行的优化修改，即**在 max 层上设置评估下限值α**（代表可能步骤的最大下界），**在 min 层上设置评估上限值β**（代表可能步骤的最小上界）。

最大下界α：作为 MAX 节点，假定它的 MIN 节点有 N 个，当第一个 MIN 节点的评估值为α时，则对于后面的节点，如果有高于α的节点，就取最高的节点值作为 MAX 节点的值；否则，该节点的评估值为α。所以当发现后面节点的值比α还小的时候，这个节点树枝就可以停止搜索了。

最小上界β：作为 MIN 节点，假定它的 MAX 节点有 N 个，当第一个 MAX 节点的评估值为β时，则对于后面的节点，如果有低于β的节点，就取最低的节点值作为 MIN 节点的值；否则，该节点的评估值为β。所以当发现后面节点的值比β还大的时候，这个节点的树枝就可以停止搜索了。

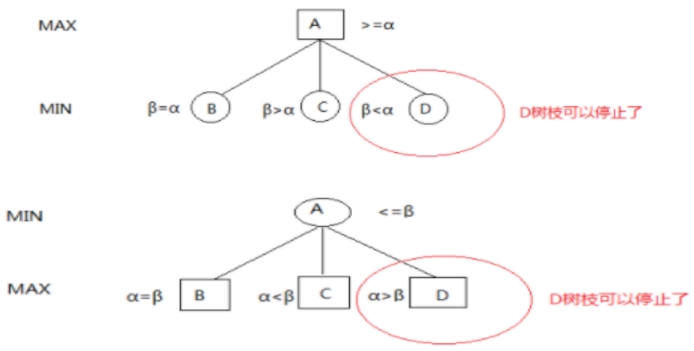

图 12 Alpha-Beta 剪枝算法示意图[6]

5.2 具体实现

代码**总体上与 Minimax 一致，添加了两个变量α和β**，分别代表可能步骤中的最大下界和最小上界，我们将其作为参数加入 maxLevel 函数和 minLevel 函数。同时加上一个判断语句，判断当前最大得分是否大于对β和当前最小得分是否小于α，若满足此条件，则直接返回该得分；若不满足此条件，则更新α以及更新β，然后返回该得分。在 max 层，如果 **maxvalue>β**，就没有继续往下走的必要了，可以直接返回 maxvalue；在 min 层中，如果 **minvalue< α**，也可以进行剪枝了，直接返回 minvalue。核心代码如下图 13、14 所示：

```
    def maxLevel(gameState, depth, alpha, beta):
        currDepth = depth + 1
        if gameState.isWin() or gameState.isLose() or currDepth == self.depth:
            return self.evaluationFunction(gameState)
        maxvalue = -999999
        actions = gameState.getLegalActions(0)

        alpha1 = alpha
        for action in actions:
            successor = gameState.generateSuccessor(0, action)
            maxvalue = max(maxvalue, minLevel(successor, currDepth, 1, alpha1, beta))
            if maxvalue > beta: #最大得分超过beta
                return maxvalue
            alpha1 = max(alpha1, maxvalue)
        return maxvalue #返回最大得分
```


图 13 Alpha-Beta 剪枝 maxLevel 函数实现

```
def minLevel(gameState, depth, agentIndex, alpha, beta):
    minvalue = 999999
    if gameState.isWin() or gameState.isLose():
        return self.evaluationFunction(gameState)
    actions = gameState.getLegalActions(agentIndex)
    beta1 = beta
    for action in actions:
        successor = gameState.generateSuccessor(agentIndex, action)
        if agentIndex == (gameState.getNumAgents() - 1):
            minvalue = min(minvalue, maxLevel(successor, depth, alpha, beta1))
            if minvalue < alpha: #最小得分小于alpha
                return minvalue
            beta1 = min(beta1, minvalue)
        else:
            minvalue = min(minvalue, minLevel(successor, depth, agentIndex + 1, alpha, beta1))
            if minvalue < alpha: #最小得分小于alpha
                return minvalue
            beta1 = min(beta1, minvalue)
    return minvalue
```


图 14 Alpha-Beta 剪枝 minLevel 函数实现

其 getAction 函数跟 Minimax 中的 getAction 函数的**唯一不同**就是增加了 alpha 和 beta 这两个变量，alpha 和 beta 的初始值分别为**负无穷**和**正无穷**，以便后续的更新迭代。

[6] 参考：https://developer.aliyun.com/article/49996

```
actions = gameState.getLegalActions(0)
currentScore = -999999
returnAction = ''
alpha = -999999
beta = 999999
for action in actions:
    nextState = gameState.generateSuccessor(0, action)
    score = minLevel(nextState, 0, 1, alpha, beta)
    if score > currentScore:
        returnAction = action
        currentScore = score
    if score > beta:
        return returnAction
    alpha = max(alpha, score)
return returnAction
```
@xxf

图 15　getAction 函数实现

5.3 结果展示

实验手册表明：“在 smallClassic 布局中**深度为 3** 的树的理想情况下，剪枝后每次移动的运行耗时**不慢于几秒钟**”。我们接下来将分别**使用 AlphaBetaAgent 和 MinimaxAgent 在 smallClassic 地图下深度分别为 2 和 3 的情况下进行 10 次运行**，以验证 AlphaBetaAgent 是否有显著提升。

(1) AlphaBetaAgent，depth=3 :使用命令 *python pacman.py -p AlphaBetaAgent -a depth=3 -l smallClassic -q -k 1 -n 10*，耗时为 **83.52s**（单次确实不慢于几秒钟）

```
The time consumed is: 83.66539096832275 s
Average Score: 779.3
Scores:        410.0, 131.0, 537.0, 699.0, 1231.0, 884.0, 1096.0, 1249.0, 943.0, 613.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```
@xxf

图 16 Alpha-Beta 测试结果（*depth=3*）

***(2) AlphaBetaAgent，depth=2*: 耗时 9.65s:**

```
The time consumed is: 9.651278018951416 s
Average Score: -105.4
Scores:        -480.0, -138.0, -396.0, -276.0, -184.0, -456.0, -72.0, 710.0, 454.0, -216.0
Win Rate:      2/10 (0.20)
Record:        Loss, Loss, Loss, Loss, Loss, Loss, Loss, Win, Win, Loss
```
@xxf

图 17 Alpha-Beta 测试结果（*depth=2*）

***(3) MinimaxAgent，depth=3*：耗时 113.29s:**

```
The time consumed is: 113.2893340587616 s
Average Score: 729.8
Scores:        887.0, -59.0, 398.0, 1089.0, 253.0, 978.0, 1246.0, 597.0, 984.0, 925.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```
@xxf

图 18 MinimaxAgent 测试结果（*depth=3*）

***(4) MinimaxAgent，depth=2*：耗时 13.61s**

```
The time consumed is: 13.610228776931763 s
Average Score: -205.9
Scores:        -540.0, -138.0, 792.0, -238.0, -459.0, -444.0, -158.0, -158.0, -464.0, -252.0
Win Rate:      2/10 (0.20)
Record:        Loss, Loss, Win, Loss, Win, Loss, Loss, Loss, Loss, Loss
```
@xxf

图 19 MinimaxAgent 测试结果（*depth=2*）

将上述 4 个实验结果汇总得到表 x，可以直观看到在 Alpha-Beta 剪枝算法下，**算法效率相比于未改进的 Minimax 大大提升，而且深度为 2 和 3 的搜索时间也变得更加接近了。**

表 4 AlphaBetaAgent 和 MinimaxAgent 实验耗时对比

Type	Depth=2	Depth=3
MinimaxAgent	13.61s	113.29s
AlphaBetaAgent	9.65s	83.66s

但实验手册所说的：“也许深度为 3 的 Alpha-Beta 树的访问会跟深度为 2 的 Minimax 树一样快”,从实验结果看**似乎还是有很大的差距**（在 *n=10* 的情况下，是 83.66s 和 13.61s 的差距）我试想这种情况，应该只有在**运行一次且很巧合**，即 MinimaxAgent 刚好运行时间相对较慢，而 AlphaBetaAgent 运行时间相对较快的情况下才会发生。经过多次实验后，也确实验证了这个猜想，从图 20 可以看到，在 *n=1* 的情况下，***MinimaxAgent，depth=2*** 耗时 ***4.56s***，而 ***AlphaBetaAgent，depth=3*** 耗时 ***5.26s***,确实已经**接近一样快**了。

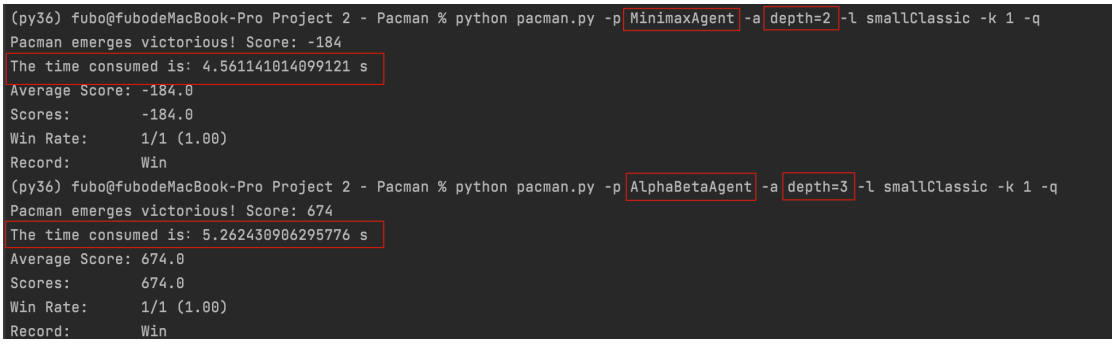

图 20 *MinimaxAgent depth=2* 和 *AlphaBetaAgent depth=3* 测试结果

6 评价函数(题 4)

题 3 的实验是在**内置的评价函数**下的 Alpha-Beta 智能体测试结果，可以看到胜率和得分都比较低，因此，我们有必要去**优化**评价函数。

与题 1 的评价函数不同的是（Reflex 智能体评价的是下一步行动），这道题的评价函数应**评价当前状态**，对于 Alpha-Beta 新智能体而言，我们需要**自底向上**对节点进行评价得分，以便**自顶向下**的高效搜索。对于评价得分的函数，直接使用题 1 的评估函数，即综合考虑了**当前状态下**吃豆人和所有**豆子的距离**、和不同状态下**幽灵的距离**、以及和**胶囊的距离**。

在代码实现方面，*betterEvaluationFunction()*函数跟题 1 的评估函数是基本一样的。只是在运行时，需要将 AlphaBetaAgent 类中 *return self.evaluationFunction(gameState)*语句改为我们自己定义的评估函数 *return better(gameState)*。

接下来我们进行测试，通过命令 *python pacman.py -p AlphaBetaAgent -a depth=2 -l smallClassic -q -n 10* 运行，结果如下图 21 所示，在深度为 2 的参数下，运行 10 次的**平均得分为 1217.1，获胜率为 100%，提升了无数倍！**说明了在按不同权重考虑了豆子的曼哈顿距离、不同状态幽灵的曼哈顿距离以及胶囊的曼哈顿距离后，我们的评价函数是很有效的。(同样的参数，如图 17 所示，使用内置函数的**平均得分为-105.4，获胜率为 20%**）

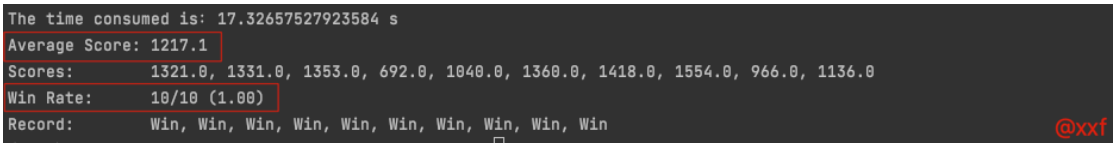

图 21 修改评价函数后的实验结果图

四、实验结论或体会

1、本次实验是在一个 **Python 工程包**中写代码，因此需要调用各个文件中的函数和类属性，这就需要熟悉每个文件中的函数和类的设计。本次实验的工程包设计十分专业，第一次

尝试读懂这么多文件一起运行的程序，并且熟练调用各个文件中某些类的成员函数，也提高了读懂代码并实现的能力，也学习到了如何拆解一个工程，使其简洁易读。

2、本次实验具有一定的**挑战性**，不仅需要由之前实验打下的编程基础，还需要熟练掌握对抗搜索算法并将其应用。虽然最后比较完整地做完了，但期间也遇到的很多的问题，也是查找了很多资料，看了很多技术贴之后才理解吸收并完成实验的。从开始到做完，花了近一周的时间，经历了好几个熬夜。

3、本次实验也十分具有**趣味性**，实验的完成也是由简单到复杂循序渐进的。通过一步步改进，最终使得吃豆人变得更加智能。不仅学习到了很多知识性的东西，而且感受到了那种强烈想把一件事情完成的动力感和充实感。同时也感受到了其实实现某个功能还是蛮难的，需要自己进一步去查找资料，理解和思考问题，最后才根据自己的理解去完成实验。

4、总的来说，通过本次实验，我学会了评估函数的设计，Minimax 算法的实现、Alpha-Beta 剪枝的实现，以及比较全面地了解了博弈对抗问题的解决方法。对于评估函数的设计与优化，能将胜率提高到 **100%**，以及平均得分提高到 **1263 分**。Minimax 智能体的胜率为 **64%**左右，以及 Alpha-Beta 剪枝策略满足了实验要求。

五、思考题

设计的 AI 博弈程序一般的优化方法有哪些？

（1）减少搜索范围。针对待解决问题的特点，制定减小搜索空间的策略。对于吃豆人游戏可以不考虑全地图的食物、药丸和幽灵，例如只考虑以吃豆人为中心边长为 4 的正方形内的对象。在地图很大地情况下，可以有效地提高运行速度。

（2）设置博弈风格。为使 AI 博弈程序更加智能，我们可以通过调整一对关键系数的比值进行博弈风格的调整。

（3）利用多线程。可以利用多核技术，让算法在多个线程下并行计算，提高速度。

（4）增大搜索层数。但搜索层数并非越大越好，尤其是幽灵的水平不如吃豆人的时候，反而会出现浪费，搜索层数应当根据计算机的性能进行调整。

（5）使用置换表。为了避免重复搜索已经搜索过的结点，从而加快搜索效率，可以使用一张表格记录每一结点的搜索结果。

（6）启发式搜索。通过构建某些特殊状态“定式”并存入数据库中，类似于置换表，若发现当前状态存在于数据库中，则直接按照数据库进行下一步。吃豆人游戏在某些困境中会存在一些提高生存几率的走法，这些走法作为启发式函数，可以让吃豆人直接按照这些走法进行移动。

（7）自学习。为了防止 AI 重蹈覆辙，在每次 AI 输的时候进行回退，然后再往下进行预测，找到不回导致失败的局面，然后记录下该局面，从而达到自学习的效果。